CHAPTER

4

ALLOCATING AND MANAGING OPERATING SYSTEM RESOURCES

After reading this chapter and completing the exercises, you will be able to:

- ♦ Explain what applications, processes, and threads are and how they fit together
- ♦ Understand how different server types have different priorities
- ♦ Explain how memory management works
- ♦ Understand the role of the Object Manager in Win2K
- ♦ List the kinds of objects found in Win2K
- ♦ Explain how Win2K object-level security works

Modern operating systems such as Windows 2000 (Win2K) are multitasking, which means that more than one application can run at the same time. (In single-tasking operating systems like DOS, only one application can run at a time.) If you look at the taskbar in your Win2K computer, you'll likely see a number of application icons representing running applications. Many applications are running simultaneously—for that matter, many parts of the operating system are running simultaneously—and these contending tasks must all compete for CPU time so that they can carry out their duties. One fundamental problem that any operating system must address is how to allocate system resources among all applications running on the computer—not to mention the various parts of the operating system itself.

For any multitasking operating system, resource management is key to its success. If resource management is done well, then applications and the parts of the operating system that need to process data will work together to make it appear that the computer is doing many things at once. If resource management is done badly, then tasks don't finish, applications hang, and users get cranky. This chapter discusses the mechanics of how Win2K manages resources to make the operating system run as smoothly as possible, allocating resources where they're needed without starving less important functions. To manage these resources, Win2K represents them as objects for the convenience of the parts of the executive component that must manipulate them.

DISTRIBUTING CPU CYCLES

The central processing unit (CPU) is the brain of the computer and is its most important part. Everything else in the computer—disks, memory, cards, and so on—is either storage or a means of getting data to the CPU for processing. Once the CPU processes the data, something happens: letters appear on the monitor, a file is saved to disk, or data are moved from the paging file to **physical memory**. In all cases, however, the CPU must be involved.

Note

Some computers with multiprocessing capabilities contain more than one CPU, implying that no single processor is central. (If more than one processor in a computer is running Win2K, then the processors are equal in status; under normal circumstances, one isn't preferred to the other.) For this reason, you'll sometimes hear the CPU in a machine referred to as the processor. Both terms refer to the same piece of the computer.

The operating system controls how that CPU time is distributed among the applications and the operating system. The problem is that the CPU can do only one thing at a time. Even in multitasking operating systems, which may have dozens of applications running at the same time, the CPU can crunch numbers for only one of those applications at any given moment. Thus the CPU's time is divided into **cycles**, which are discrete chunks of time that the CPU can dedicate to any given application's needs. The trick is to distribute those cycles in such a way that the computer seems as responsive as possible, giving the most time to applications running in the foreground without starving the applications running in the background.

Recall from the discussion of the Process and Thread Manager in Chapter 2 that Win2K doesn't really recognize applications as such. When you run an application, the Win2K Process and Thread Manager creates a **process**, which defines the resources available to that application. Among these resources are the following:

- The importance of this process (and thus of the threads within it) relative to the other processes running on the computer.
- The virtual memory addresses in which the threads belonging to this process can store data.
- In multiprocessor systems, the preferred processor that threads in that process should use, if any.
- The location of the page directory for virtual memory dedicated to that process. The page directory is required when translating virtual memory addresses to physical memory addresses.

Note

If a process is set up to prefer using one processor over another, then the process is said to have an **affinity** for that processor. Normally, the threads in a process use whichever processor is least busy. In some cases, however, the person who wrote the application may direct the process to have an affinity for a certain processor, so that threads belonging to that process will run only on that processor unless explicitly directed otherwise, even if another processor is less busy.

The process also contains at least one **thread**, which is the executable part of the application. That is, a thread is responsible for carrying out a task. The system contains a thread for accepting keyboard input, a thread for saving data to disk, and so forth. The Process and Thread Manager creates threads only as they're needed to complete a task; it then deletes each thread as its task is completed and returns the resources used by that thread to the process's pool of resources. Basically, the application is the user's front end to all threads that are running and to the process that defines how those threads will get storage space and CPU cycles. You interact with the application; Win2K interacts with the processes and threads that make the application do whatever it does.

As you can see, then, the problem lies not in getting CPU cycles to applications, but in getting them to the threads that support the tasks that the applications are supposed to complete. That's a somewhat complicated chore, and the subject of the following section.

Understanding Multitasking Types

Resource management in a multitasking operating system such as Win2K is geared toward making it *appear* to the user as though all programs are running concurrently without interfering with each other. Strictly speaking, running multiple applications simultaneously isn't possible from the point of view of Win2K. The CPU can do only one thing at a time, so some form of multitasking is needed.

There are three main kinds of multitasking in the Windows world. A simple multitasking environment performs **task switching**, in which multiple applications may be open at a single time, but only the **foreground application**—those being used at the moment—gets any CPU cycles. The **background applications**—those opened but not receiving input—stop working until they're in the foreground again.

A more even-handed approach is **cooperative multitasking**, which is used in Windows 3.x. This mechanism gives all applications access to the CPU for a set period in round-robin format. The "cooperative" part comes from the fact that the applications are supposed to cooperate with each other and voluntarily relinquish the CPU when their time for using the CPU is up. It doesn't always work this way, however. A misbehaving application or an application performing an especially complicated calculation may not relinquish the CPU, thus freezing all other applications in place until the misbehaving application comes to its senses or you restart the machine.

To avoid this problem, Win32 operating systems, including Win2K, use **preemptive multitasking**. This method basically takes control of the CPU away from the applications and gives it to a part of the operating system. Rather than being dependent on the goodwill of the running applications to relinquish the CPU to other applications, preemptive multitasking can interrupt a running application to give CPU time to an application currently running in the background, even if it's not currently getting user input.

Thread Scheduling Basics

Win2K uses a priority-driven preemptive multitasking system to allocate CPU time to threads. Thus the thread with the highest priority that's ready to run always gets the CPU. If more than one thread has the same priority, then those threads share CPU time among themselves in round-robin fashion, so that they all get equal time. If a thread with a certain priority is running and a thread with a higher priority becomes ready to run, then the thread with the higher priority displaces the running thread (more precisely, Win2K displaces the running thread) and gains control of the CPU. The threads that are ready to run may belong to the same process or to different processes. Scheduling happens at the thread level and is unrelated to the processes, except in terms of how process priority affects thread priority.

When a thread is ready to run, it continues for a preset number of CPU cycles called a **quantum**. A quantum is the length of time that a thread is allowed to run before Win2K interrupts the thread to see whether any other threads with the same priority are ready to run. The quantum isn't consistent for all threads; as you'll see, the number of cycles in a quantum can vary from thread to thread and also depending on whether you're running Windows 2000 Server or Windows 2000 Professional. No Registry setting controls quantum length. Instead, each thread has a quantum value that represents the number of CPU cycles in its quantum. Threads running under Windows 2000 Server have a longer quantum than threads running under Windows 2000 Professional. The reasoning behind this difference is as follows: Any task you ask a server to perform is probably urgent and should be completed all at once. The longer quantum is intended to make it more likely that the task will be completed in a single quantum.

Whenever a thread begins running, the CPU must load that thread's **context**, or the information defining its operating environment. What priority is the thread? In which addresses can it store data? What resources does it have available to create other threads or processes, if necessary? This information is recorded in the thread's context. When the CPU loads a new thread's context into memory, this procedure is called **context switching**.

Note

If two threads from the same process get the CPU in succession, then no context switching is necessary. The information is already loaded.

As you may recall from Chapter 2, the part of Win2K that takes responsibility for scheduling the threads lies in the kernel. The kernel does not contain a single scheduling mechanism. Rather, the code is spread through all parts of the kernel that interact with threads, with the code segments collectively being called the kernel's **dispatcher**. The kernel's dispatcher has much the same function as the dispatcher for a taxi company: it keeps track of all units (taxicabs or threads) and organizes them.

The dispatcher is triggered by any of the following events:

- A thread becomes ready to run.
- A thread stops running because it has finished its quantum, the thread terminates, or the thread waits for other input without which it can't continue.

- A thread's priority changes.
- In multiprocessor systems, a running thread's **processor affinity** changes.

In any of these cases, Win2K must determine which thread should next get the CPU. As mentioned earlier, it will be the thread of highest priority that's ready to run on that CPU.

Note

If no thread is ready to run, then the idle thread gets the CPU. The idle thread isn't really a thread, but more of a placeholder—the cybernetic equivalent of the CPU twiddling its thumbs while waiting for something to do. The idle thread isn't perfectly idle, as it keeps an eye on the system to see whether any real work becomes available, but it doesn't do anything itself.

Understanding Thread States

A thread may not be ready to run because of its **thread state**. A thread may be in any of the following states:

- *Ready:* A ready thread isn't waiting on anything and is, as the name implies, ready to execute. The dispatcher considers only ready threads when looking for threads to give CPU cycles.
- *Standby:* The thread has been selected to run next on a particular processor. When the running thread stops, the dispatcher performs a context switch to this thread. Only one thread may be in the standby state for each processor in the computer—essentially, it's the next thread in line.
- *Running:* Once the dispatcher has performed the context switch to load a thread's process information, the thread starts its quantum and gets CPU time. It continues to run until one of the following events happens: its quantum ends, the dispatcher stops the thread to run another thread with a higher priority that's entered the ready state, the thread terminates, or the thread voluntarily enters the wait state because it needs more information.
- *Waiting:* A waiting thread has been running but is now on hold, either waiting for more data or having been pulled from the running state to let another thread execute. When a thread exits the waiting state, depending on the circumstances it may either resume running or return to the ready state, where it may be chosen to run.
- *Transition:* A thread is ready to run but some of its kernel resources have been paged out of memory so that the CPU can't execute the thread's task due to incomplete information. Once a thread exits the transition state, it is in the ready state.
- *Terminated:* A thread has finished executing. The Process and Thread Manager may delete this thread or, if it might be needed again, may keep it around and reuse it in the same process later.

A thread's life cycle might look like the following example: A process has created Thread Alpha, with a priority of 8. Thread Alpha is now ready to run. When the dispatcher examines the threads in the ready state, it picks Thread Alpha as having the highest priority and puts

Alpha into the standby state for the processor, poised to take over the CPU when the currently running thread finishes its quantum. The running thread terminates, and Alpha enters the running state, getting all of the CPU's time.

But wait! Thread Beta, with a priority of 12, jumps into the fray. This thread has just entered the ready state. The dispatcher, which is responsible for ensuring that the CPU is always busy with the most important task available, puts Alpha into the standby state and gives the CPU time to Beta. When Beta finishes its quantum, the dispatcher examines the available threads. Because Alpha is in the standby state and no threads with a higher priority are ready, the dispatcher lets Alpha go back to the running state, where it finishes its quantum.

When Alpha finishes its quantum, it also completes its job, so the dispatcher puts Alpha into the terminated state. From here, the Process and Thread Manager can either delete Alpha or keep it around in case it needs another thread object. This situation implies a tradeoff. On the one hand, keeping the thread object uses up some of the resources allocated to the process to which the thread is related, while making those resources unavailable to other threads that the process might need to create. On the other hand, it saves time if the process needs a new thread, because it doesn't have to go through the (fairly lengthy) process of creating a thread when it needs one.

Thread and Process Priority

Priority is the key to how threads get CPU time. Consequently, it's important to understand how priorities are set and how they work. When a process is created, it's assigned a priority that reflects its importance. Each thread that's created in the context of this process has a priority influenced by, but not necessarily identical to, its process's priority. A thread's starting priority—its **base priority**—matches that of its process. Win2K may raise or lower that priority for some threads under certain conditions.

Most threads are **variable priority threads**, which means that Win2K can raise or lower their priorities as circumstances dictate. Exceptions include the important system threads that belong to a different class of processes, called real-time processes, which have a higher priority than all user application threads. Any thread with a real-time priority level keeps its original priority.

Note

Although Win2K has such "real-time" processes, this terminology is something of a misnomer. Win2K is not a real-time operating system because it has no way to ensure that a thread executes at a certain time—a requirement of a true real-time operating system. Win2K uses thread priority to do its best to ensure that important threads run as often as possible, but that is the limit of its ability.

Applications are designed to run at a given priority level, usually normal. You can tweak the priority for user applications, however, if you start them from the command prompt. Using the start command, you can make an application run as a high-priority or even real-time priority process. For example, to run Solitaire in real time, you would type start /realtime sol. You can also change process priorities from the Task Scheduler. There is a catch: if you run applications in real time, Win2K essential services may be less responsive, because more threads will contend for CPU time.

Scheduling Scenarios

So far, you have learned about the possible states of threads and the dispatcher's scheduling of threads to run based on their relative priority. You will now look at how this system can work in practice, based on a few possible scenarios:

- Quantum end
- Voluntary switching
- Preemptive thread scheduling
- Thread termination

Quantum End Recall that every thread running under Win2K has a quantum based both on the thread and on the version of Windows 2000 you're running—Server or Professional. When the running thread finishes its quantum, Win2K must decide whether its priority should be reduced and, if so, whether another thread should run.

Why would Win2K reduce a thread's priority? The operating system will sometimes raise a thread's priority to ensure that it receives CPU time—the priority-driven scheduling makes Win2K vulnerable to starving low-priority threads for CPU cycles. Once the thread has had some time to run, Win2K may lower its priority back to its original level.

If Win2K lowers the thread's priority, the dispatcher runs the thread currently in the standby state, if no other thread with a higher priority has become available in the meantime. The thread that had been running goes to the back of the line of threads that are in the ready state and that share the same priority, as shown in Figure 4-1.

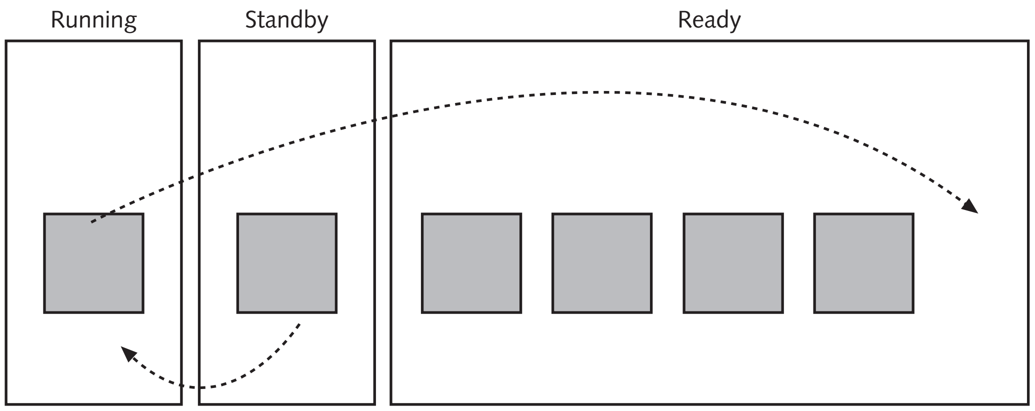


Figure 4-1 When a thread has finished running, it goes to the back of the line of ready threads of the same priority

Voluntary Switching Just because a thread has a quantum doesn't mean that it will always finish that quantum. A thread might leave the running state and enter the waiting state, either voluntarily or when preempted by the thread dispatcher.

If a thread is waiting for something else to happen, then it will voluntarily enter a waiting state, as shown in Figure 4-2. This situation might happen, for example, if the thread needs some I/O to complete before it can complete its own task. A thread isn't penalized for its "generosity" in relinquishing the processor. If it gives up the processor to another thread that's ready to go, then the waiting thread receives a priority boost when it's ready again so that it gets the processor immediately (instead of having to wait for the thread to which it yielded the processor to complete its quantum). It will also be allowed to restart its quantum.

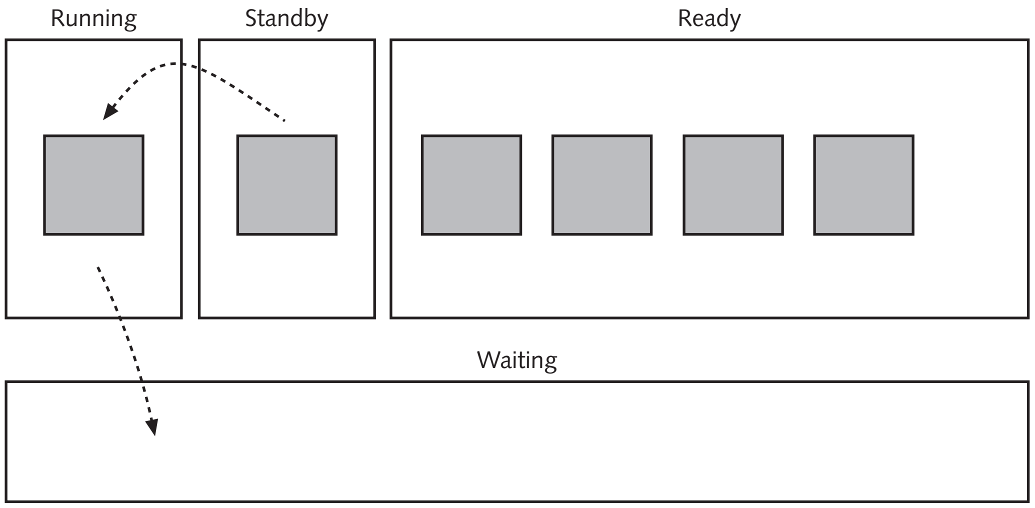


Figure 4-2 A thread waiting for something will go into a waiting state and yield the CPU to the next waiting thread

Preemption Even if a thread doesn't wait voluntarily, it may end up waiting anyway. If a thread is running and a thread with higher priority becomes available, then the running thread will be preempted and the higher-priority thread will get the CPU. This situation works somewhat differently than voluntary switching. First, as you can see in Figure 4-3, the preempted thread does not enter a waiting state. Rather, the preempted thread enters the standby state so that when the higher-priority thread finishes, it will be next in line. Its priority does not rise; instead, it merely waits for the higher-priority thread to finish so that it can resume its quantum. Second, the preempted thread does not restart its quantum, but rather completes the quantum it had already started.

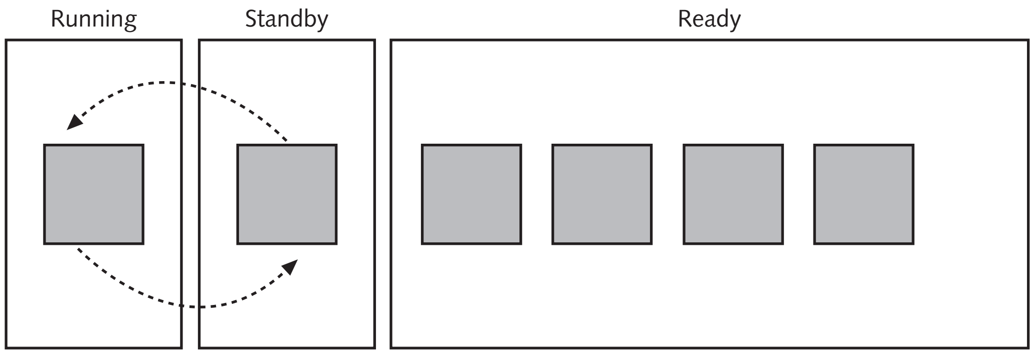


Figure 4-3 A thread preempted by another thread will enter the standby state until the second thread finishes its quantum

Termination As explained earlier, when a thread finishes its task (not just its quantum), it goes into the terminated state, as shown in Figure 4-4. If the thread object is no longer needed by any running process or thread, then it is deleted and its resources go back into its process's resource pool. If the thread isn't deleted, its priority remains unchanged, but it won't get CPU cycles until it's reinitialized and sent back to the ready state.

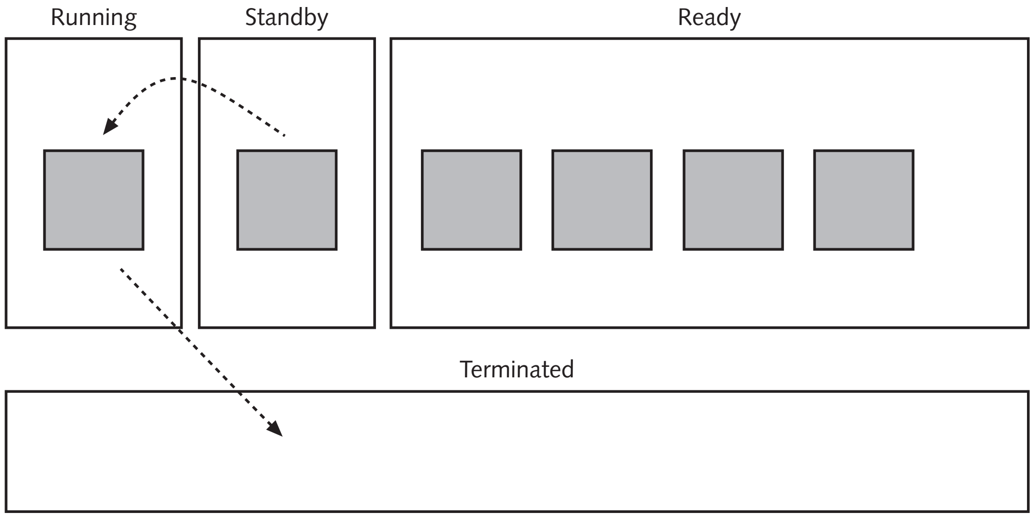


Figure 4-4 A thread that finishes its assigned task enters the terminated state

Adjusting Thread Scheduling

To make the system more responsive, Win2K may edit thread priority or extend a thread's quantum. These adjustments may include any of the following:

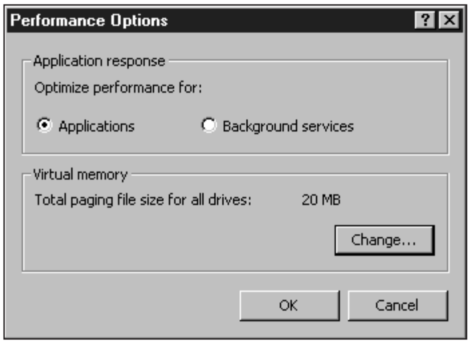
- Boosting priority for foreground applications
- Boosting thread priority for threads that have stopped waiting for interrupts to be processed

- Boosting thread priority for threads entering a waiting state
- Boosting thread priority for threads not getting CPU time

Priority Boost for Foreground Applications The simplest way that Win2K can ensure that a thread gets enough time is to give it a slight priority boost. Depending on how Win2K is set up, the foreground application may receive a slight or not-so-slight priority boost.

You can set up your Win2K computer to run applications more efficiently (this strategy is recommended for Windows 2000 Professional or for Windows 2000 Servers supporting terminal session clients). With a system configured in such a way, user applications running in the foreground get a priority boost of 12 when they're accepting input, then drop to a priority of 11 at intervals when they're in the foreground but not immediately receiving user input. (Unless you're copying something previously written, it's unlikely that you'll be constantly inputting data into an application.) If you've set up the Win2K computer as a server, so that it gives more or less equal time to all applications, the foreground application *still gets a priority boost*, albeit a smaller one than it would on a machine optimized for running applications. In this case, the foreground application's priority is boosted from the usual 8 to 10 while the application is actively accepting input, but drops to 9 periodically if you stop entering data into the application. In either case, as soon as another application enters the foreground, the first application's priority reverts to its base priority of 8. (Priority settings range from 1 to 15.)

To edit or check the setting, open the System applet in the Control Panel and access the Advanced tab. If you click the Performance Options button, you'll see a dialog box like the one shown in Figure 4-5. If you set the application response to optimize performance for running applications, then the foreground application gets a longer quantum; if you set it for background services, then all applications have the same quantum. This information is recorded in the Registry under HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Priority Control\Win32PrioritySeparation. If you change the setting, it's reflected immediately in the Registry—you do not have to restart the computer.


Performance Options

Application response

Optimize performance for:

[✓ Applications] Background services

Virtual memory

Total paging file size for all drives: 20 MB

Figure 4-5 Options for treating foreground applications

Priority Boosting After Waiting for I/O Operations If a thread has gone into a waiting state because it needs some I/O operation (such as reading from a disk) to complete, then Win2K will give that thread a priority boost when the operation is finished and the thread is ready to run again. As a result, the threads will run, complete what they were doing, and finish their quantum so that other threads can run.

The actual degree of priority boost depends on how long the thread had to wait for the I/O operation to finish. With a short wait, the thread gets a small boost in priority; with a long wait, it receives a larger boost. Thus threads waiting on relatively fast devices, such as locally accessible CD drives or hard disks, will experience a smaller priority boost, and threads waiting on slower devices, such as the keyboard, will have a much larger one. The idea is that the priority should increase in line with the performance hit that the thread experienced while waiting. The actual priority is determined by the device driver used to access the I/O device. Thus the dispatcher doesn't evaluate the length of the wait, but rather the device on which the thread was waiting. For example, a thread that had been waiting for the end of an I/O operation that depended on the hard disk driver would get a temporary priority boost of 1. A thread waiting on keyboard input would receive a temporary priority boost of 6.

Note

Only threads with priorities in the variable range will get a priority boost. They'll also be boosted only up to the top of the variable range. In other words, a priority boost will not suddenly give a variable thread real-time priority. For example, if a thread has a base priority of 8 (normal for user applications), a priority boost of 8 will give it a priority of 15, which is the top priority for variable priority threads. A thread with a base priority of 15 would receive no boost at all, because its priority is already as high as it can go.

After its priority is boosted, the thread can run one quantum at the increased priority level. When that quantum ends, the dispatcher reduces that thread's priority by one level—to use the usual jargon, decrements it by 1. If the thread is still the highest-priority thread available, then it gets another quantum, after which it's decremented again. This process, which is illustrated in Figure 4-6, continues until the thread reaches its original priority.

Priority Boosts After Waiting for User Input or User Messages Users are a lot slower than disks. To help threads waiting for user input before they can continue what they were doing, the Win2K dispatcher raises the priority of threads waiting for user responses or for window messages once the threads have received this input and are ready to start again. It also doubles the next quantum for that thread, so that the thread has a chance to finish the task that required user input.

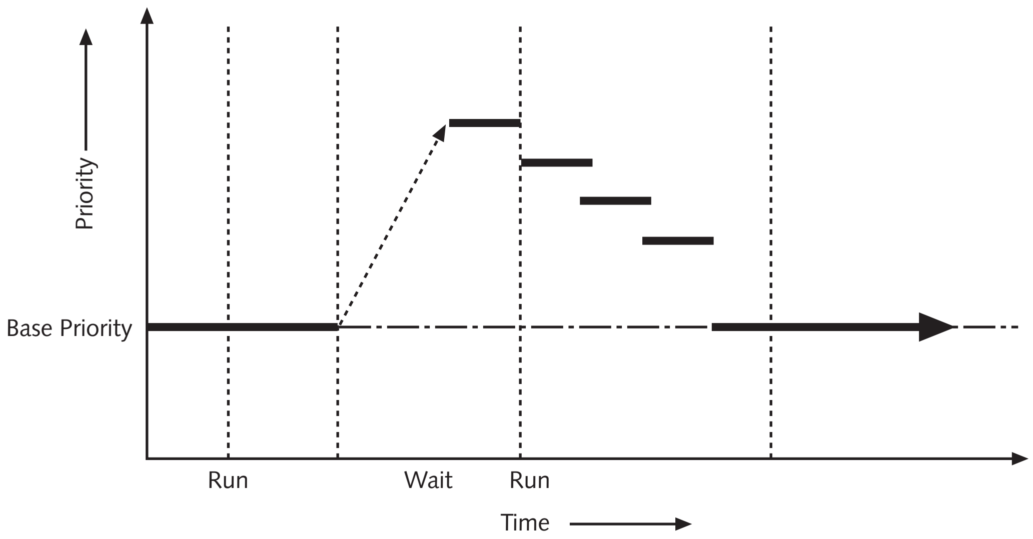


Figure 4-6 The dispatcher raises the priority of waiting threads to an amount based on the device for which they were waiting

As shown in Figure 4-7, this priority boost works somewhat differently from the priority boost that threads can gain after their I/O-based wait ends. First, it is always the same amount. Second, it doesn't last as long as the I/O-based boosts do, because the boost caused by waiting for user input persists for only a single quantum. When that quantum has finished, the thread drops back immediately to its base priority.

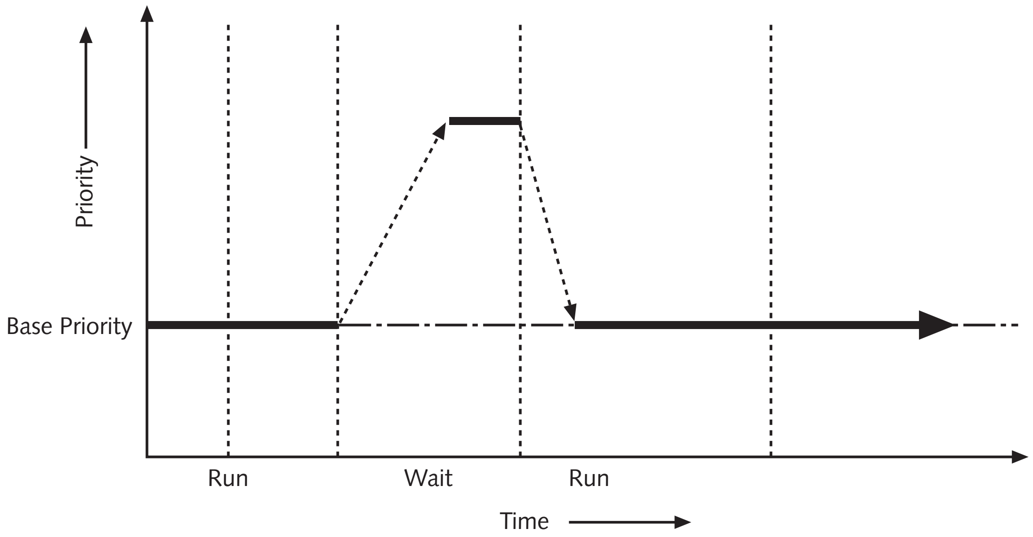


Figure 4-7 Threads waiting for user input get a single big boost after the wait ends, then drop to their base priority

Priority Boosts to Rectify CPU Starvation Many threads are interdependent. It's perfectly possible—and quite common—for one thread to depend on output from another thread's execution.

But consider the following: A thread with a priority of 12 is running, keeping a thread with a priority of 8 from getting any CPU time. A thread with a priority of 14 needs the priority-8 thread to finish, however, before it can complete its own task. Although the priority-14 thread should be able to bump the priority-12 thread, it can't, because the priority-8 thread must finish first. The priority-4 thread will never finish, however, because the priority-12 thread is getting all of the CPU time while the priority-14 thread remains in the waiting state.

To resolve this situation, a part of Win2K called the **balance set manager** must become involved. Although the balance set manager is mostly concerned with memory management functions, it has one scheduling trick: It scans the queue of ready threads, looking for those that have been ready to run but haven't been able to do so for longer than a few seconds. If it finds such a thread, the balance set manager will boost the thread's priority to 15—the highest priority available to a variable priority thread—and give it double the normal quantum. Effectively, this action not only gives the starved thread immediate control of the CPU, preempting the priority-12 thread that originally caused the trouble, but also gives the starved thread a longer period to complete its task. Once this long quantum ends, the thread's priority reverts to its original base priority. If the two quantums weren't sufficient to allow the thread to finish its job, then the thread returns to the ready queue, waiting for the balance set manager to notice that it's not running and to give it another doubled quantum. Figure 4-8 illustrates this scenario.

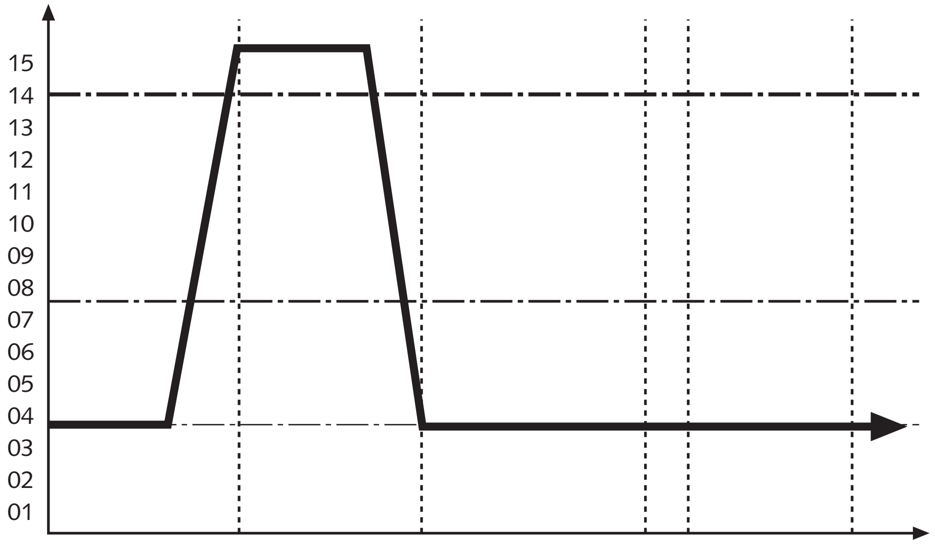


Figure 4-8 The balance set manager gives CPU-starved threads both a high temporary priority and a doubled quantum

This mechanism won't always immediately resolve bottlenecks caused by high-priority threads that depend on low-priority ones. Eventually, however, it will get the low-priority threads out of the way so that the threads dependent on them can execute.

CPU time is merely one resource that Win2K must control to make applications run well. Another resource is virtual memory, which is controlled by the Virtual Memory Manager (VMM).

MEMORY MANAGEMENT

Even in these days of servers with 1 GB of physical RAM installed, no server has enough memory to keep up with the demands of all applications running on it. Therefore, Win2K uses disk space to emulate physical memory, essentially "faking out" the applications and operating system by storing only the data they currently need to operate; the rest is stored in the paging file, a location on the hard disk. When the application needs data in the paging file, Win2K brings it back into physical memory. To the application, it appears that the data was always stored in RAM, with the only difference being that, if the data appeared in the paging file, its retrieval takes a bit longer. This emulation allows all active processes to think that they have 4 GB of memory in which to store data (4 GB is the maximum amount of memory that Windows 2000 will support).

As you can see in Figure 4-9, this 4 GB is divided into two halves: system space (kernel) and per-process space (virtual memory). The kernel is the part of the operating system that supports items such as boot drivers, the system cache, and other elements that every process will need. It claims the upper 2 GB of memory. This part of the **virtual memory** address space is shared among all running processes. The lower 2 GB of virtual memory is reserved for user-specific applications and is private to each process.

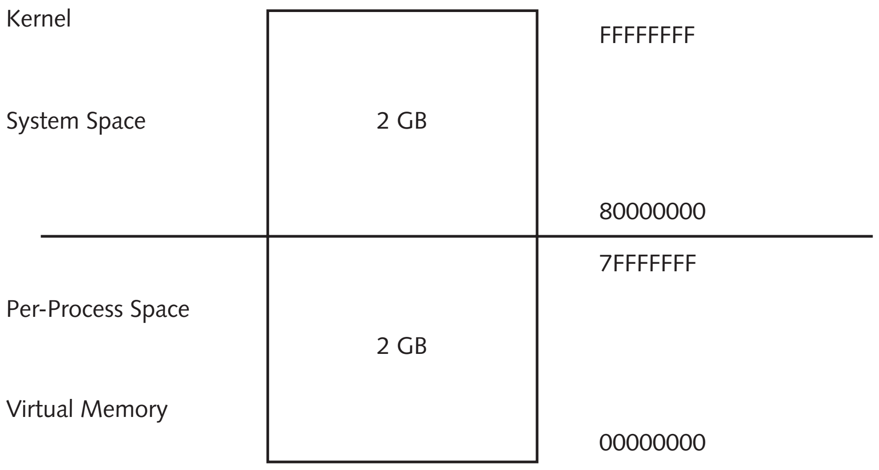


Figure 4-9 Distribution of virtual memory addresses

Virtual address space is merely a list of storage spaces available to the threads running within a process, like an inventory sheet for a warehouse. The physical organization of the warehouse may not map precisely to the inventory sheet, but so long as everyone knows that Entry 5 on the inventory sheet corresponds to Bay 20 in the warehouse, it doesn't matter. The Win2K memory manager takes charge of keeping track of the process's inventory lists so that they don't try to store data in the same location in physical memory.

Note

Win2K includes support for terminal services, a way of letting network clients run applications on the server while seeing the application interface on their computer. Because terminal services imply that many people are running applications on the same computer, Win2K needs a way to keep these applications sorted out so that it can tell which applications are associated with each person. Each client connection is called a session. Each time a remote user connects to a Win2K terminal session, a new session ID is generated. All of the processes created to run within that session are associated with that particular session ID. This approach keeps Win2K from becoming confused as to which process in which session asked for what data.

Keeping track of how virtual memory and physical memory are used is a big job. To achieve this goal, you need a device that will send data to disk when it's not being used, but then retrieve the information when the process refers to that data again. That is, you need a method of translating the virtual memory addresses to the physical memory where the data is actually stored. You need some mechanism for reserving memory for certain processes and for preventing processes from corrupting each other's memory. All these jobs are handled by a part of the Win2K executive called the Virtual Memory Manager.

Earmarking Memory for Processes

When a process is created, Win2K must allocate some virtual memory addresses to that process for its exclusive use. The operating system supports a two-stage approach to memory allocation. First, processes can reserve free (unused and unreserved) memory addresses. Second, they can commit that allocation by backing those addresses with some storage. You can't put anything in an address any more than you could live in a new housing development that consisted of only empty plots with addresses attached to each plot. To live there, you'd need to build a house. Likewise, to store memory at an address, a process must have storage to which that address refers.

Reserving memory is simply a way for a thread to say, "I'd like to use these virtual memory addresses at some point in the future." **Reserved memory** doesn't take up any room in the paging file or in RAM, because the process isn't actually using the memory. Before the process can store memory in that range of addresses, it must commit it. **Committed memory** reserves a space in the paging file for that process's exclusive use.

Why bother with this two-step approach? Two reasons explain its attractiveness. First, it's handy for processes that need a large chunk of virtual memory addresses next to each other—they can reserve them all at once and then commit later. Second, this method can reduce paging file usage by not committing any space in it to the virtual memory addresses until this space is actually needed.

Memory is reserved—and therefore committed—in multiples of the **system page** size. (A page is an area of memory; all processors deal with memory in pages, with the size of the page varying with the processor type.) For example, on x86 systems, processes must reserve memory in 4 KB increments, because x86 systems process memory requests in 4 KB chunks. If a process attempts to reserve 18 KB of virtual memory addresses, the actual amount reserved would be 20 KB because the amount of space used must be a multiple of 4.

Sharing Memory Among Processes

Sometimes it makes sense for processes to share data. First, processes may share even process-specific data, rather than loading identical copies of the same material into memory. Second, under normal circumstances, all processes share a common view of the lower 2 GB of virtual memory addresses.

Copy-on-Write Protection

When you launch an application, you actually load certain data into memory that the threads must reference to do their work. For example, if you run WordCruncher 98, files A through E are loaded into memory to provide basic functionality. If you start a second instance of WordCruncher, is it really necessary to load all of those files into memory again? You could, but it would consume a lot of memory. If you let all instances of WordCruncher use the same copies of A through E, however, then one instance that needs to overwrite part of that information will affect all other instances.

To circumvent the wasted space/data corruption dilemma, Win2K uses a technique called **copy-on-write data sharing**. Basically, it means that if more than one application is referencing data on a read-only basis, as many applications as need to and are able to may reference the same data loaded into the same part of memory. If an application writes to that data, however, the Virtual Memory Manager will copy the edited data to a new location.

Protecting System Memory

Although all processes share a common view of the lower 2 GB of virtual memory addresses, you still need some mechanism to protect the data there, thus ensuring that a process (or even the operating system), doesn't have its data corrupted. Win2K includes some measures to prevent such corruption.

First, only kernel-mode threads can access data in the system's virtual address areas. As discussed in Chapter 2, when a user-mode thread needs to manipulate the kernel, it doesn't actually perform the manipulation itself. Rather, the executive accepts the request and uses a kernel-mode thread to interact with the system memory areas. User-mode applications cannot directly modify or even access system data.

Second, threads don't actually interact with virtual memory addresses—the Virtual Memory Manager handles that task for them. When a thread tries to read or write the contents of virtual memory, the Virtual Memory Manager intervenes to handle the address translation. In that way, Win2K ensures that threads belonging to one process don't accidentally access memory pages belonging to another process.

Third, memory areas are protected. Memory areas may have different permissions attached to them (making some read-only and some read-write, just like disk space). Shared memory areas are protected with normal Win2K access control lists that control access to all objects. When a thread attempts to access an area of virtual memory, the Security Reference Monitor in the executive checks the permissions associated with that thread against the permissions required to access that memory. If the thread doesn't have the appropriate permission set, access is denied.

Using the Paging File

For virtual memory to work, you need a paging file—that is, an area of hard disk space reserved for data that processes load into memory. Committed memory is memory that has space in the paging file reserved for it. Processes can't use RAM without having a location to which to transfer their data if RAM becomes too full.

Unfortunately, no matter how much RAM is installed in the machine, it will become full sooner or later. Every time you run an application, you start at least one process. That process, in turns, starts threads. All of those threads need some data in memory—collectively called the process's **working set**—to work. Once you start loading user files into memory as well, things get crowded very quickly.

You may recall that the balance set manager was responsible for making sure that low-priority threads got some CPU time. This same balance set manager has another job: making sure that only the data processes need most actually takes up room in physical memory. If physical memory gets too full, the balance set manager **trims** each process's working set, sending nonessential data to the paging file. Which data? The Win2K Virtual Memory Manager uses one of two algorithms for this kind of operation: **Least Recently Used (LRU)** or **First In, First Out (FIFO)**. The LRU algorithm discards the data that's been in memory for the longest time without being used; FIFO discards the data that's been in memory for the longest time, regardless of when it was last used. Multiprocessor computers and all Alpha-based computers use FIFO, whereas single-processor x86 computers use LRU. LRU is somewhat more responsive to the needs of the user and the operating system, but FIFO is faster, because it doesn't require the Virtual Memory Manager to determine which data was least recently used.

Reading Data from Memory

Now you know how data gets into memory—but how does it get out when a process needs to use some of that information? The Virtual Memory Manager acts as an intermediary between application threads and the physical memory, both for security reasons and because virtual memory addresses don't necessarily have any relationship to the location in RAM where the data is actually stored. To retrieve data from memory, the Virtual Memory Manager must perform **address translation** to convert the virtual memory addresses into physical memory addresses.

Virtual addresses are not mapped directly to physical ones—no direct correlation exists between virtual address C80000 and physical memory address 65, for example. Instead, virtual memory addresses are associated with a structure stored in a non-pageable part of system memory, known as a **page table entry (PTE)**. The PTE contains the physical address to which the virtual address is mapped. Mapping addresses via the PTE intermediary means that if the data is paged to disk and then back to a different area of RAM, the virtual memory address need not be updated to reflect the change in location—only the PTE requires updating.

Of course, the procedure is actually more complicated. If the Virtual Memory Manager had to sort through 4 GB of PTEs to find the one that translated a virtual memory address into a physical memory address, it would take forever. For this reason, PTEs are organized into **page tables**, and the page tables are gathered into **page directories**. Essentially, this setup is analogous to the drive/folder/file structure that file systems use to help you organize your data.

Each virtual memory address is 32 bits long. Although you normally see these addresses written in hexadecimal for ease of use, it's easier to understand how address translation works if you write the address in the binary format that the operating system uses internally. For example, imagine that a thread has asked for the data in virtual memory address 8000A500. In binary notation, this address is written as 10000000000000001010010100000000.

The reason for writing out this address the long way is that virtual memory addresses are interpreted as three separate components: 1000000000 0000001010 010100000000. Counting from the left, the first 10 bits of the virtual address point to the proper page directory. The second 10 bits point to the proper page table in that page directory. The final 12 bits point to the byte of data within the page.

Using this information, the process of translating memory addresses is as follows:

1. The Virtual Memory Manager finds the page directory for the current process. The current process is the one that owns the thread that's currently using the CPU; the location of the page directory is part of the thread's context and is therefore loaded into memory when the thread starts running.
2. The Virtual Memory Manager reads the page directory to find the page table that holds the required PTE.
3. The PTE is read to map the virtual address to a page in physical memory.
4. The byte index is used to find the desired data within the physical memory. If the data currently resides in memory, the Virtual Memory Manager retrieves the information for the thread.

If the data *isn't* currently in memory, the Virtual Memory Manager executes a **page fault** to find the data in the paging file and send it back to physical memory. The **page fault handler** searches the paging file for the desired data (all of which is cataloged so it can be easily retrieved) and puts it back into physical memory.

Note

The page fault handler puts not only the requested data back into memory, but also the data around it. It assumes that if the application is using one part of data, it might need the data close to it as well.

If physical memory is full, then the balance set manager evaluates process data and trims the working set according to the appropriate algorithm. The PTE will be updated to reflect the new physical address, and the Virtual Memory Manager can retrieve the data for the thread.

By this point, you should have a good idea of how Win2K manages CPU time and virtual memory. The brains behind both operations is the Object Manager, which is the subject of the next section.

OBJECT MANAGER

The core of all resource management in the Win2K is the Object Manager, which creates and destroys the objects that represent system resources, thus allowing all resource management to be performed from a single point in the operating system instead of making each part of Win2K handle its own resource management. Basically, each kind of object represents a specific kind of computing resource, user data, or part of the computer. For example, a file object represents the data in a user file, a device object might represent a floppy disk drive, and a process object represents a process.

The environmental subsystems in which you run applications interact with objects to gain access to resources in the same way that you use a mouse to move the insertion point across the screen. You do not actually move the insertion point: in fact, even the mouse doesn't actually move the insertion point. Instead, the mouse is a controllable representation of the internal mechanism that moves the insertion point.

Microsoft designed the Object Manager with the following goals in mind:

- To provide a consistent mechanism for managing system resources
- To manage all object security from a single point in the operating system
- To supply some mechanism for charging processes for the resources they use
- To allow one process to inherit resources from another process that created it
- To supply a mechanism for keeping an object available until all processes are finished with it

Recall from Chapter 2 that the core operating system contains two main parts: the executive and the kernel. The executive runs in user mode, giving the environmental subsystems access to core operating system functions and making policy decisions about who is allowed to do what in the operating system. The kernel runs in kernel mode and synchronizes the parts of Win2K to make them cooperate.

To keep consistent with this division of labor, Win2K has two kinds of objects: executive objects and kernel objects. The parts of the executive (Process and Thread Manager, Virtual Memory Manager, I/O Manager, and so on) use executive objects, which include policy information. Kernel objects are implemented within the Win2K kernel and provide the timing mechanisms that Win2K uses to keep the parts of the executive working together.

Executive Objects

Executive-mode functions need executive objects to give the functions access to system resources or parts of the computer. Executive objects are the user-mode representations of kernel-mode object functions, so an executive object may actually supply executive functions with access to a number of kernel functions. The kernel functions, in turn, may create executive objects while using kernel objects to keep track of how those executive objects are being used. For example, imagine that you're running Microsoft Word in the Win32 environmental subsystem and you tell Word to create a new file. Word asks the Win32 subsystem to create the file (remember—applications can't actually do anything; they just ask the operating system to do it for them). The Win32 subsystem takes the request and calls on the executive-level function CreateFile. The executive, in turn, verifies that this request is permitted and then passes it to the kernel-mode function that creates files. The kernel creates an executive-level file object that the Win32 operating system can use to manipulate the data in that file.

Figure 4-10 illustrates the three parts of an executive object: the labeling information that the object manager uses to keep track of the object; the kernel component; and the executive component. As you can see, when part of the operating system manipulates an executive object, it automatically manipulates the kernel object nested within the executive object, even if that part of the operating system is unaware of the kernel objects.

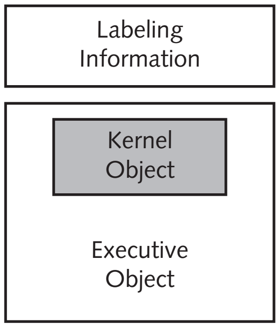


Figure 4-10 An executive object containing a kernel object

Many kinds of executive objects exist. You don't need to know the names of all of them, but some of the more important ones will come up in the discussion of how Win2K manages resources. A thread, as discussed in Chapter 2, is an executable part of an application that has a certain job to do. All threads in the same application are part of one process that describes the conditions under which the threads run: virtual address areas available to them, priority, and so forth. Processes communicate with each other via ports. All processes and threads have **access tokens** that contain their security profiles. Each profile consists of an identifying name and a set of rights and permissions based on the rights and permissions of the person who initiated the action that led to the process being created initially (for example, starting an application).

As discussed earlier in this chapter, all threads in all processes contend for CPU time, so some timing objects are also important. Mutants are responsible for ensuring that only one thread at a time accesses a resource that can't handle multiple simultaneous requests. For resources that can handle more than one simultaneous thread request, but not an unlimited number, semaphores allow only a certain number of threads to reach the resource. A timer tells a thread when a specified period of time has elapsed.

Calling All Objects!

Most Win2K objects have names that other objects (such as process objects) can use to get their attention. (Objects without names can't be called and are very rare.) The Object Manager also uses these names to keep track of objects. When a process creates an object or refers to an existing object by name, the process receives a **handle** to that object. It's faster for a process to refer to an object by its handle than by the object's real name. To understand why, consider this analogy: once a process has a handle to an object, it's like having a red telephone on that object's desk. The process dials the red telephone, and the object is connected to the process instantly. The Object Manager, which would normally have to find the object for the process, never becomes involved. Because it's so much more efficient to call a process via its handle than through the Object Manager, threads in a process cannot communicate with an object until that process has established a handle with the object.

Because the Object Manager is responsible for creating and deleting all object handles, it can keep track of which user-mode processes are trying to access objects and determine whether the security profile of that process permits the kind of access to the object that the process is seeking. This concept leads us to object security.

Setting Object Security

When you log onto a Win2K computer, the Security Reference Monitor compares your user name and password with the security database contained on the Win2K domain controller. Based on the rights and privileges associated with your user name, you—and the objects you create—have a certain set of rights, represented by an access token attached to the object. When an object you create attempts to access another object, the Security Reference Monitor checks the **access control list (ACL)** of the object, which lists who's allowed to do what to the object. For example, a file object's ACL might permit members of the Everyone group to read the file, but restrict members of the Account Operators group to editing the file. The Security Reference Monitor compares the data in the ACL with the information in the access token. If the access token lists the appropriate permissions, then the object gets the access. If it doesn't, then access is not allowed and you receive an Access Denied error.

Chapter Summary

- In any multitasking operating system, the biggest headache lies in making sure that each part of the operating system gets enough CPU time and memory to be responsive, but not so much that other parts that also need those computing resources are starved.
- This chapter explored how Win2K divides applications and the operating system itself into discrete parts, then assigns resources to each of those parts.
- This chapter also discussed how Win2K uses objects to represent computing resources and how it manages security for those resources.

Key Terms

access control list (ACL) — A list associated with an object that defines the rights that groups and individuals have to the object. The ACL is used by the Security Reference Monitor to protect objects from unauthorized access.

access token — An identifier given to an object upon its creation. Based on the identity of the person who created it (or who created the object that created it), an object has certain rights, which are listed in its access token. The Security Reference Monitor compares the data in the access token with that required by the ACL to determine what kind of access the object may have to a particular object.

address translation — The act of converting virtual addresses to physical addresses. This conversion is necessary because the operating system deals entirely in virtual addresses, leaving physical memory addresses to hardware. The two types of addresses don't necessarily bear any relation to each other.

affinity — The term used when a process is set up to prefer using one processor over another.

background application — An application that is running but not currently receiving user input.

balance set manager — The part of Win2K responsible for trimming process working sets to free physical memory as well as for identifying low-priority threads that aren't receiving CPU cycles.

base priority — The priority with which a thread starts after its creation. The base priority of a thread is always equal to that of the process that created it.

committed memory — Memory allocated to a process that is backed with the necessary amount of space in the paging file. Processes must commit memory before they can store data in it.

context — The information describing the operating environment for all threads in a particular process.

context switching — The act of setting aside one thread's context for that of another thread, when the second thread starts using the CPU.

cooperative multitasking — A type of multitasking in which all applications in turn get some CPU time and are supposed to relinquish the processor when their time is up.

copy-on-write data sharing — A form of shared memory protection. Copy-on-write allows multiple processes to read the same bit of data stored in physical memory. If one of the processes attempts to change the data, however, the Virtual Memory Manager copies the edited data to a new location and the process uses the copy. This approach keeps the editing process from corrupting the data that other processes are using.

cycles — Discrete chunks of time that the CPU can dedicate to any given application's needs.

dispatcher — A set of routines in the Win2K kernel that governs thread scheduling.

First In, First Out (FIFO) — An algorithm that marks the oldest data in RAM to be sent to the paging file. The balance set manager uses this algorithm on Alpha and multiprocessor x86 computers.

foreground application — The application currently receiving user input.

handle — A connection to an object that allows one object to manipulate another.

idle thread — A low-priority thread that runs whenever no other threads are running on the CPU. The idle thread watches for events that will require CPU time, but doesn't actually do anything with the CPU itself.

Least Recently Used (LRU) — An algorithm that marks the least recently used data in RAM to be sent to the paging file. The balance set manager uses this algorithm on single-processor x86 computers.

page directory — A collection of page tables for a particular process.

page fault — An event in which the Virtual Memory Manager must retrieve data from disk to put it back into RAM for a process.

page fault handler — The part of the Virtual Memory Manager that finds the data that's been paged to disk so as to put that data back into RAM.

page table — A list of page table entries, used to map virtual addresses to storage areas in physical memory.

page table entry (PTE) — The entry on a page table that contains the mapping of physical storage to virtual memory addresses.

physical memory — The memory chips installed in the computer that are used for temporary storage of process data. Synonymous with random access memory (RAM).

preemptive multitasking — A type of multitasking in which the Virtual Memory Manager controls who has control of the CPU, rather than giving this responsibility to the applications.

process — The environment defining the resources available to threads, which are the executable parts of an application. Processes define the memory available, any processor affinities, the location where the process page directory is stored in physical memory, and other information that the CPU needs to work with a thread.

processor affinity — In multiprocessor systems, a feature that may be used to tell all threads in a process that they should use one processor in preference to another, even if the preferred processor is busier than the alternative processors.

quantum — The number of CPU cycles that a thread gets to use when executing. During its quantum, a thread gets all of the CPU's attention.

reserved memory — Virtual memory addresses set aside for a particular process but not yet committed—that is, no space in the paging file has been reserved for them.

system page — Chunks of memory, as viewed by a processor. The system page for an x86 machine is 4 KB in size; for an Alpha machine, it is 8 KB in size.

task switching — A method of multitasking in which the user may switch between applications. The application in the foreground gets all CPU cycles; the background applications get none.

thread — The executable element of an application.

thread state — Any one of five states that a thread may be in, defining its readiness to use the CPU.

trim — The procedure in which some of a process's working set is moved to the paging file to free room in physical memory.

variable priority thread — A thread with a base priority from 1 to 15 that may have a higher priority if the dispatcher thinks it appropriate. A variable priority thread may never have a priority higher than 15.

virtual memory — A mechanism by which RAM is supplemented with disk space to make it appear that the computer has more memory installed than it really does.

working set — The data that the threads in a process have stored in physical memory. The working set may grow or shrink depending on how much physical memory is available, but the process may not use any data that is not in its working set.

Review Questions

1. Win2K is a(n) ________________ operating system; DOS is a(n) ________________ operating system.
2. A processor in a computer running a multitasking operating system such as Win2K can execute more than one application at a time. True or False?
3. If the threads in a process are designed to prefer one processor over another (in a multiprocessor computer), then those threads are designed to have a(n) ________________ that processor.
   a. preference for
   b. affinity for
   c. association with
   d. None of the above
4. In any multitasking operating system, the application in the foreground is the one that receives most of the CPU cycles. True or False?
5. Describe how an application in a cooperative multitasking operating system can cause all applications to hang, and explain how a preemptive multitasking operating system can prevent this event from happening.
6. What is a quantum?

7. What is context switching?
   a. The act of loading one process's environment information into memory and setting aside the environment information of another process
   b. The act of moving from kernel mode to user mode, or vice versa
   c. The act of shifting working set data from main memory to the paging file
   d. The act of changing a thread's priority to help it gain CPU cycles
8. Which part of Win2K holds the dispatcher?
   a. Executive
   b. Win32 subsystem
   c. HAL
   d. Kernel
9. When a thread is scheduled to be run next on a particular CPU, but is not presently running, then the thread is said to be in the ________________ state.
10. A preempted thread goes into the waiting state. True or False?
11. What is the usual base priority of a thread associated with a user application?
    a. 5
    b. 8
    c. 12
    d. 24
12. Explain why Win2K is not a true real-time operating system.
13. Threads that voluntarily relinquish the processor get a boost in priority once they're ready to start running again. True or False?
14. If you configure a Win2K machine to give all applications equal access to the CPU, then the foreground application does not receive a priority boost. True or False?
15. A thread awaiting an I/O operation exits the waiting state when the I/O operation is complete. How is the priority boost given to this thread determined?
16. Which part of the Virtual Memory Manager is involved both in scheduling threads and in conserving system RAM?
17. When old data in a process's working set is sent to the paging file, this is known as ________________ the working set.
18. Explain how a faster hard disk could improve virtual memory performance.
19. What does a PTE do?
20. When a process marks a set of virtual memory addresses for its own use, it is said to be ________________ this memory. When the process backs these addresses with space in the paging file, those addresses are said to be ________________.

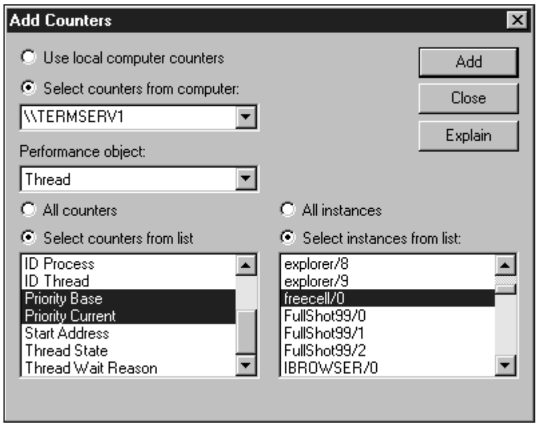
HANDS-ON PROJECTS

Project 4-1

Win2K boosts the priority of the foreground application to make it as responsive as possible. The amount of the boost depends on whether the Win2K machine is optimized for running applications or for serving client requests. In either case, however, the active application is boosted. In this exercise, you'll optimize the server for running applications and then watch the thread associated with that application have its priority raised and lowered.

To watch a thread's priority change as its application moves from foreground to background:

1. Make sure that the computer is optimized for running applications. Open the **Control Panel** and run the **System** applet. Access the **Advanced** tab, and click the **Performance Options** button you see there. You should see a dialog box like the one shown earlier in Figure 4-5.
2. Close the **Performance Options** dialog box.
3. Run **FreeCell**, a game in the **Games** folder of the **Accessories** menu. (This application has only a single thread.)
4. Run **Performance Monitor** (**Start**, **Run**, **perfmon**). Click the button with the + sign on it to add a new counter.
5. Scroll down the list of performance objects until you find the **Thread** object. Scroll down in the list of counters until you see the counters for **Priority base** and **Priority Current**. Hold down the **Ctrl** button, and click the counters to select both. In the list of instances, choose **FreeCell**. There should be only one thread for that application. When you finish making the selections, your screen should look like Figure 4-11. Click **Add**, then click **Close** to close this dialog box and return to the chart.
6. Right-click the graph numbers on the left of the chart and select **Properties** from the resulting menu.
7. Select the **Graph** tab and change the Maximum value for Verticle scale to **20**.
8. Click **OK**.


Add Counters

Use local computer counters
[✓ Select counters from computer:] \\TERMSERV1

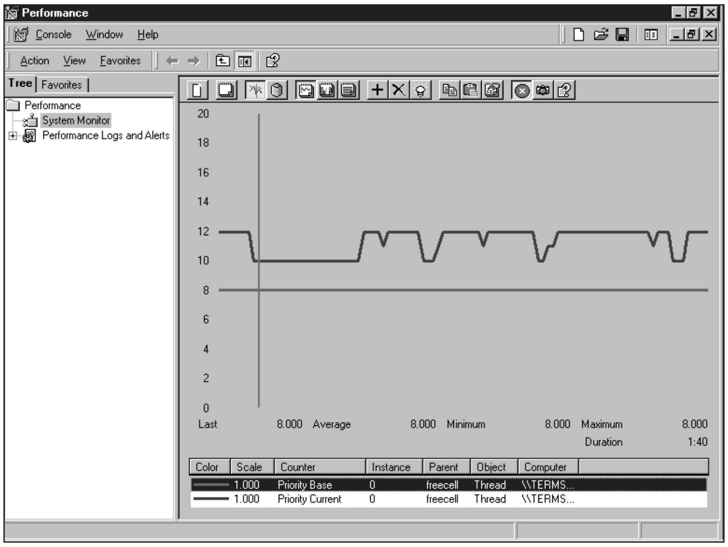
Performance object: Thread

All counters
[✓ Select counters from list]
- ID Process
- ID Thread
- Priority Base (selected)
- Priority Current (selected)
- Start Address
- Thread State
- Thread Wait Reason

All instances
[✓ Select instances from list:]
- explorer/8
- explorer/9
- freecell/0 (selected)
- FullShot99/0
- FullShot99/1
- FullShot99/2
- IBROWSER/0

Figure 4-11 Choose to display the current and base priority for the FreeCell thread

9. Experiment with FreeCell. Notice that when you're actively clicking on the game, its priority is 12. When you click **Performance Monitor** to move that application to the foreground, but leave FreeCell up and open, FreeCell's thread priority drops to 10. When you minimize the game, the thread's priority should drop to 8. Figure 4-12 shows the changes in thread priority.
10. Close all open applications.



Last 8.000 Average 8.000 Minimum 8.000 Maximum 8.000
Duration 1:40

Color	Scale	Counter	Instance	Parent	Object	Computer
▬	1.000	Priority Base	0	freecell	Thread	\\TERMS...
▬	1.000	Priority Current	0	freecell	Thread	\\TERMS...

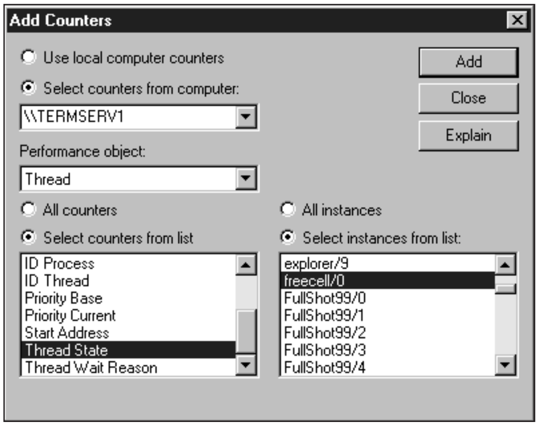
Figure 4-12 Observe how the thread's priority changes depending on how you're interacting with the game

Project 4-2

You can also use Performance Monitor to see the state of a thread at any given time. Just as the application you have open is idle more often than not, a thread is waiting for input more often than it's running. Although the Performance Monitor's documentation of the Thread State counter is slightly off, you can still use it to determine how much time an active application's threads are running as opposed to waiting for input.

To determine how much time a thread spends waiting as opposed to running:

1. Open **FreeCell**.
2. Open **Performance Monitor**, click the **Add** button (the one with a plus sign on it). As in Project 4-1, scroll down the list of performance objects until you come to the **Thread** object. In the list of Thread counters, find **Thread State**. In the list of objects to monitor, find FreeCell. When you've made all of the selections so that your screen looks like the one in Figure 4-13, click **Add**, and then click **Close**.



Add Counters

Use local computer counters [✓ Select counters from computer:]

\\TERMSERV1

Performance object: Thread

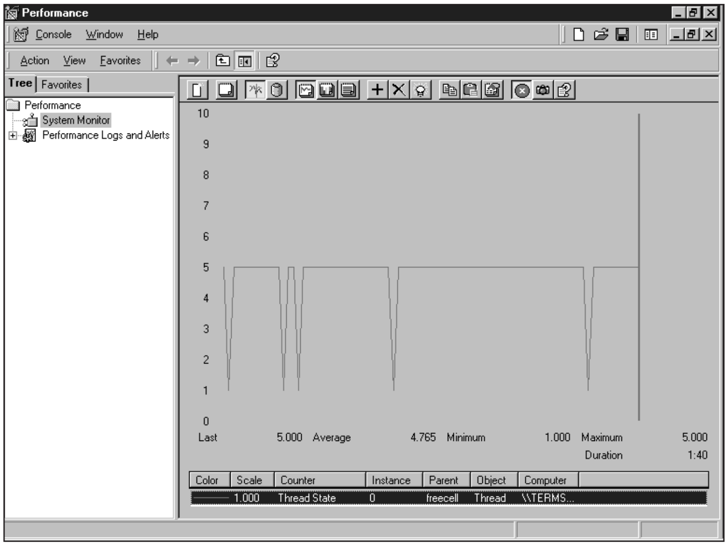
All counters [✓ Select counters from list]

All instances [✓ Select instances from list:]

ID Process	explorer/9
ID Thread	**freecell/0**
Priority Base	FullShot99/0
Priority Current	FullShot99/1
Start Address	FullShot99/2
Thread State	FullShot99/3
Thread Wait Reason	FullShot99/4

Figure 4-13 Make these selections to monitor the state of the FreeCell thread

3. Repeat steps 6 through 8 in Hands-on Project 4-1.
4. With **Performance Monitor** still open, start playing FreeCell. Watch how the value of Thread State flips between 1 (running) and 5 (waiting). The performance chart should look like the one in Figure 4-14. Notice that the game is spending much more time waiting for user input than running.
5. Close FreeCell and Performance monitor.



Last 5.000 Average 4.765 Minimum 1.000 Maximum 5.000
Duration 1:40

Color	Scale	Counter	Instance	Parent	Object	Computer
———	1.000	Thread State	0	freecell	Thread	\\TERMS...

Figure 4-14 A thread in a user application generally spends more time waiting than executing

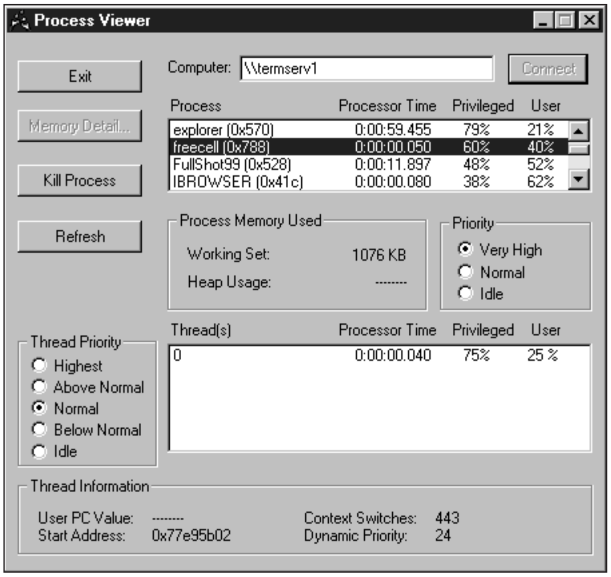
Note: Although the explanation for the Thread State counter says that a value of 1 for the thread state indicates that the thread is ready, it actually corresponds to a running thread. From time to time, you'll come across these discrepancies when using Performance Monitor.

Project 4-3

User applications are not system-critical, so they run in the normal range of priority and their threads have a base priority of 8. You can specify that an application's process—and thus the threads running within the context of that process—should have an abnormally high priority, however. To do so, you need to know the application's program name. In this exercise, you'll start FreeCell with an abnormally high priority, then compare its priority with that of a normal instance of FreeCell.

To start an application with an abnormally high priority:

1. From the **Accessories** section of the **Programs** folder, choose **Command Prompt**.
2. At the command prompt, type **start/realtime freecell**. FreeCell will start normally.
3. Run the **Process Viewer** (Pviewer) in the Win2K Support Tools, and scroll down the list of running processes to find the one for **FreeCell**. Your screen should look like Figure 4-15. Notice that the dynamic priority for FreeCell is 24 (in the real-time range).
4. Close Process Viewer.



Computer: \\termserv1

Process	Processor Time	Privileged	User
explorer (0x570)	0:00:59.455	79%	21%
freecell (0x788)	0:00:00.050	60%	40%
FullShot99 (0x528)	0:00:11.897	48%	52%
IBROWSER (0x41c)	0:00:00.080	38%	62%

Process Memory Used
Working Set: 1076 KB
Heap Usage: --------

Priority: [✓ Very High] Normal Idle

Thread(s)	Processor Time	Privileged	User
0	0:00:00.040	75%	25 %

Thread Priority: Highest Above Normal [✓ Normal] Below Normal Idle

Thread Information
User PC Value: -------
Start Address: 0x77e95b02
Context Switches: 443
Dynamic Priority: 24

Figure 4-15 If you run FreeCell as a real-time process, then it will have a higher dynamic priority than it would otherwise

Tip

You should install the Support Tools included on the Windows 2000 CD—this collection of tools can come in handy. After you've installed the Support Tools, the Programs folder will include a menu item for it.

Note

Keep this instance of FreeCell open—you'll need it for Project 4-4.

Hands-On Project

Project 4-4

In Project 4-1, you watched the priority of an application's thread change when that application was in the foreground and you were interacting with it. The dispatcher won't boost all priorities, however—just those of threads belonging to variable-priority processes. If a process has a real-time priority, then its threads will not be boosted even when you're typing into them.

To observe that only variable-priority processes change priority when in the foreground:

1. Use the command-prompt **start** command to start **FreeCell** with a high priority, if you haven't already done so.
2. In **Performance Monitor**, click the **Add** button (the one with the plus sign on it) and scroll to the **Thread** performance object. Choose the counters for the Priority Current and Priority Base, and the FreeCell instance. When your screen looks like Figure 4-16, click **Add**, and then click **Close**.

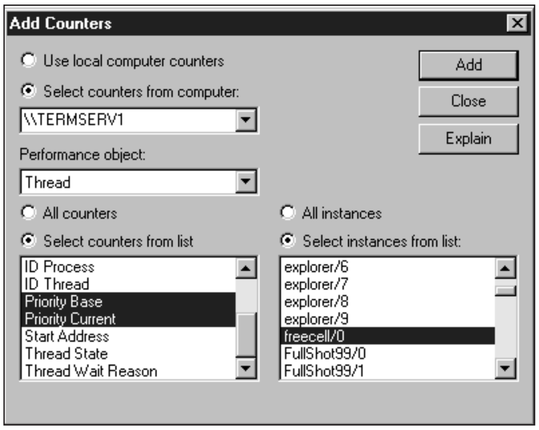
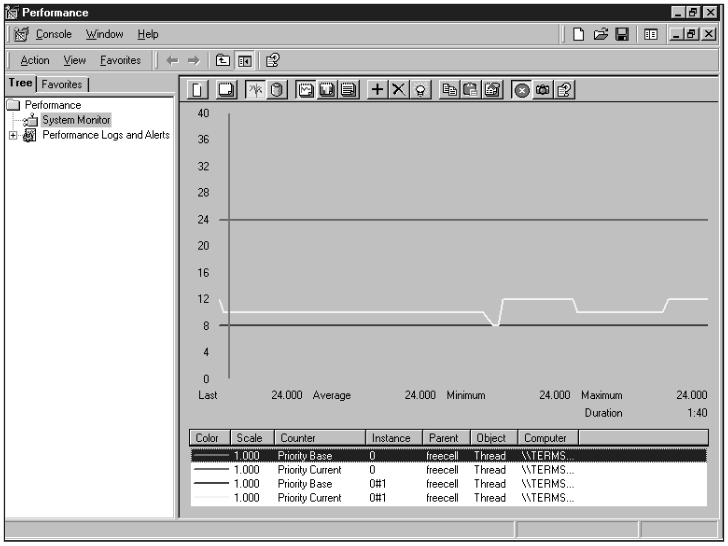


Figure 4-16 Choose the counters to monitor the FreeCell thread

3. Start **FreeCell** from the **Accessories** section of the **Programs** folder.
4. Repeat Step 2 for the new instance of **FreeCell** (it will be labeled freecell/0#1), and add the base and current priority for the normal instance of FreeCell to the chart.

5. Play with each instance of FreeCell, noticing what happens to the base and current priority of each instance of the game. As shown in Figure 4-17, you should see the priority for the normal instance of FreeCell go up and down as you use and minimize the application, respectively, but the priority of the real time instance of the game will remain constant at 24.
6. Close all open windows.



Last 24.000 Average 24.000 Minimum 24.000 Maximum 24.000
Duration 1:40

Color	Scale	Counter	Instance	Parent	Object	Computer
▬	1.000	Priority Base	0	freecell	Thread	\\TERMS...
▬	1.000	Priority Current	0	freecell	Thread	\\TERMS...
▬	1.000	Priority Base	0#1	freecell	Thread	\\TERMS...
▬	1.000	Priority Current	0#1	freecell	Thread	\\TERMS...

Figure 4-17 Only the variable priority instance of FreeCell will fluctuate in priority as it moves from foreground to background

Project 4-5

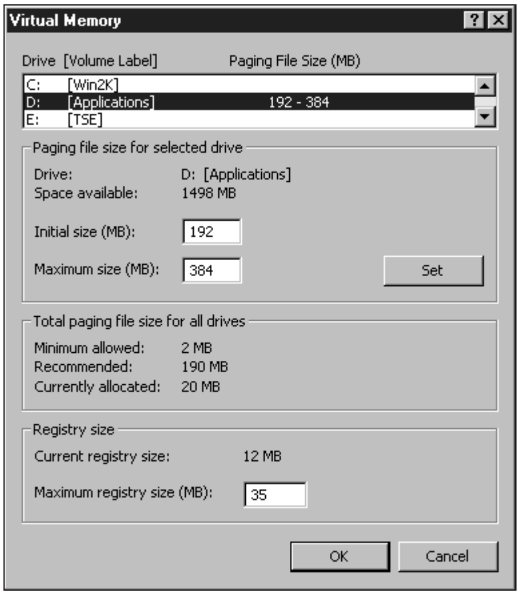
You can have more than one paging file set up on your system. Win2K will accept one paging file per logical drive (that is, per drive letter). In this exercise, you'll view and edit the paging file settings.

To view and edit the paging file information for your computer:

1. Run the **System** applet in the **Control Panel**. From the **Advanced** tab, click the **Performance Options** button to open the dialog box shown earlier in Figure 4-5.
2. The Performance Options dialog box will display the current size of the paging file. To change this amount or to see how it's distributed among logical drives, click the **Change** button to open the dialog box shown in Figure 4-18.
3. To view the paging file information for a particular drive, highlight that drive in the top box. The minimum and maximum size of the paging file for that drive will be displayed below. To change the value, type in a new minimum or maximum, being careful not to set the maximum size of the paging file smaller than the recommended size (based on the amount of physical memory installed).

Tip

Make the minimum size of the paging file close to the recommended size of the paging file, if possible. Although Win2K will extend the paging file as needed (up to the amount of room on the disk), enlarging the paging file is a time-consuming process, and one you can skip by making the paging file big initially.


Virtual Memory

Drive [Volume Label]	Paging File Size (MB)
C: [Win2K]	
D: [Applications]	192 - 384
E: [TSE]	

Paging file size for selected drive

Drive: D: [Applications]
Space available: 1498 MB
Initial size (MB): 192
Maximum size (MB): 384

Total paging file size for all drives

Minimum allowed: 2 MB
Recommended: 190 MB
Currently allocated: 20 MB

Registry size

Current registry size: 12 MB
Maximum registry size (MB): 35

Figure 4-18 The Virtual Memory settings show how the paging file is distributed on your computer

4. When you have finished making changes, click **Set**, and then click **OK** to finalize the changes.

Project 4-6

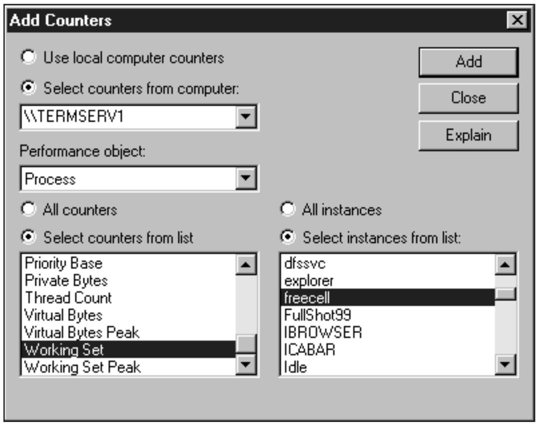
Hands-On Project

The set of data that a process maintains in physical memory is called its working set. When you're using an application, the process will attempt to build its working set as large as possible, so that it has all of the data that it might need readily at hand. If data that the process has used is not in the working set, it resides in the paging file: thus, if the process needs that data, a delay will occur while the page fault handler retrieves the data. When you're not actively using an application, there's no need for its working set to take up RAM, so the balance set manager trims the working set when applications are idle. In this exercise, you'll watch as an application builds its working set, loading as much data into RAM as it can. You'll also see how the balance set manager trims the working set when you minimize an application, then how the working set is rebuilt when you start using the application again.

To watch a process's working set change as you use the application associated with that process:

1. Make sure that **FreeCell** is running, but minimize it.

2. In the **Performance Monitor**, click the **Add** button (the one with the plus sign on it) and scroll to the **Process** performance object. Choose the **Working Set** counter and the **FreeCell** instance. When your screen looks like Figure 4-19, click **Add**, and then click **Close**.



Add Counters

Use local computer counters
[✓ Select counters from computer:] \\TERMSERV1

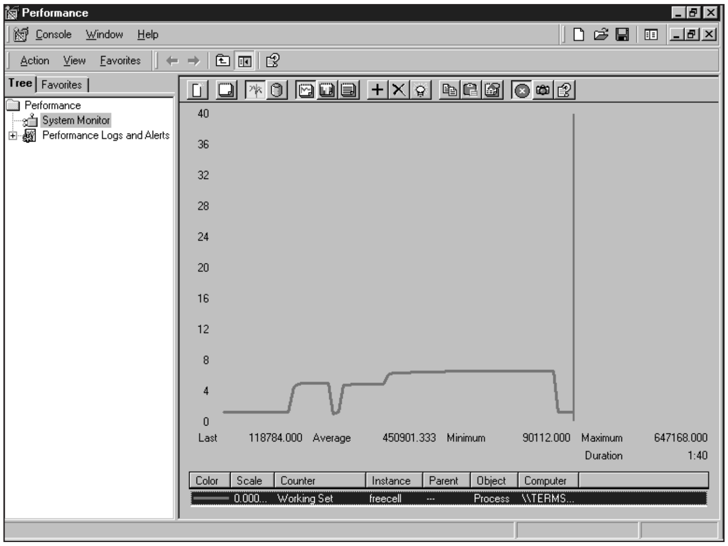
Performance object: Process

All counters
[✓ Select counters from list]
Priority Base
Private Bytes
Thread Count
Virtual Bytes
Virtual Bytes Peak
[✓ Working Set]
Working Set Peak

All instances
[✓ Select instances from list:]
dfssvc
explorer
[✓ freecell]
FullShot99
IBROWSER
ICABAR
Idle

Figure 4-19 Monitor the working set for the FreeCell process

3. Start playing FreeCell, but keep an eye on the process working set. It should start from almost nothing, then increase as you continue using the application. The longer you play the game, the larger the process's working set will be.
4. Minimize FreeCell and watch the size of the process's working set drop immediately.
5. Start playing the game again, and notice how the working set increases. Your performance chart should look something like the one in Figure 4-20.



Last 118784.000 Average 450901.333 Minimum 90112.000 Maximum 647168.000
Duration 1:40

Color	Scale	Counter	Instance	Parent	Object	Computer
——	0.000...	Working Set	freecell	---	Process	\\TERMS...

Figure 4-20 The process working set reflects how the application is being used

CASE PROJECTS

1. Your computer has been running low on virtual memory. In an effort to fix the problem, you add more physical memory. This step helps, but you still keep getting "Low on Virtual Memory" warning messages. How could you resolve the shortage of virtual memory without adding more physical memory?
2. Thread A needs information read from the hard disk to continue running, and so voluntarily gives up the CPU to Thread B, the next thread in line. What state was Thread B in, and what state does Thread A go into? When does Thread A start running again?
3. Explain the difference between task switching, cooperative multitasking, and preemptive multitasking, and explain which one Win2K uses and why.
4. Is it better for performance for Win2K to run multiple threads from the same process in succession? If it makes a difference, why?